EXTENDED RANGE AND/OR PRECISION

THE PROBLEM:

The primary stumbling block for performing shading through a multipass opengl application is the range and precision of the data types in the pipeline. The eroded shader provides a good example; it is taken from the RenderMan Companion page 353:

```
surface eroded(         float         Ks          = .4,
                                      Ka          = .1,
                                      Km          = .3,
                                      rough         = .25 )
        {
float         size  =  4.,
                       mag  =  0.,
                       i;
point         Nf, V,
                       W = transform("object",P);
point         x = (1.,0.,0.);

for ( i = 0.0; i abs(.5-noise(W*size))/size;
(b)                size  *=  2.0;
            }

            mag = mag*mag*mag*Km;

            N = calculatenormal(P-mag*normalize(N));

            Nf = faceforward(normalize(N),I);
            V = normalize(-I);

(c)             Oi = smoothstep(0.0001, 0.003, mag);

            Ci = Oi*Cs*(Ka*ambient()+Ks*specular(Nf,V,rough));
        }
```

OpenGL clamps data types to the range 0-1 at several points in the rendering pipeline. However, typical shaders will commonly exceed this range. Some examples are (letters refer to the lines indicated above):

(a) Any slowly-varying parameters will likely be interpolated as colors from the vertices. These parameters can be, such as 4.0 in this example, outside the range 0-1.

(b) Per-pixel single-component variables can have large values. In this example, size will get to 256.0 before the shader exits.

(c) Per-pixel single-component variables (and colors, such as Oi) can have small fractional values. In this example, we need to represent one ten-thousandth.

METHOD:

The ideal data type is single precision ieee floating point because of its use in unix and intel/ microsoft software renderers. Anything less and some hypothetical shader will fail where it would have succeeded with single precision ieee floating point. By the same logic there must be hypothetical shaders that fail in prman because they require double precision.

How often the failing shader will be encountered in practice is a job for analysis and experimentation. However it cannot be ruled out completely because of the simple argument above. Short of miracles, 24 bits of mantissa and 7 exponent bits plus a sign bit throughout the ball r chip is unlikely to be a reality. Yet the data paths should have as much range and precision as possible to support programmable shading in the most general way.

The choices for providing extended range and/or precision to support a shader toolkit layered on bali opengl are:

- keep unsigned, fixed point data types in the rasterizer
- introduce signed, fixed point data types in the rasterizer
- introduce signed, floating point data types in the rasterizer

We have experimented by taking sample shaders that work with ieee floating point and measuring the image degradation as the precision or range is reduced. In addition, there are a priori arguments from efficiency and complexity:

To compute z = x + y, where each value is best represented with a floating point data type but we only have unsigned fixed point available, we can attempt to use a scale to get range and a bias to get the equivalent of signed values. For example we might use a scale of .25 and a bias of .25 to simulate the range -1 to 3 with unsigned fixed point numbers. We are simply trading two bits of fraction for two integer bits; moving the binary point. With s = .25 and b = .25, to compute z = x + y we want

```
z' = x' + y'  where
x' = s*x + b,  y' = s*y + b and z' = s*(x + y) + b
```

so

```
z' = s*x + s*y + b + b
   = s*(x + y) + b + b
```

Note that z' has an additional unwanted b term. this means that for each addition in the source, another pass must be use to subtract the additional b term. If z' is to be further used in calculations, we continue after subtracting the additional b term. If z' is the final value we are interested in them we de-bias and multiply by 1/s to get z, which is a final two passes.

Addition is painful, but possible. Multiplication is probably too difficult to deal with to make it worthwhile. Consider computing z = x*y with the same scale and bias scheme:

```
z' = s*z + b
   = x'*y',
where x' = s*x + b,
      y' = s*y + b and we want z' = s*(x*y) + b.
so x'*y'  = (s*x + b) * (s*y + b)
          = s*s*x*y + s*b*x + s*b*y + b + b
```

Note that x'*y' is not s*z + b. The cleanup passes become quite complex, and the performance degrades rapidly.

If we had *signed* fixed point then automatic emulation of floating point data types in the shading language compiler becomes somewhat easier:

```
z = x + y is computed as
z' = s*z, x = s*x, y = s*y;
z' = x' + y' is just
z' = sx + sy = s(x + y).
```

This is great because now an add in the source given to the translator becomes an add in the pipe. No monkey-business with additional cleanup passes is required.

```
z = x*y  is computed as
z' = s*x * s*y
```

so an extra pass to multiply by (1/s) is required in the general case and of course a final pass is required to de-scale the result.

With signed floating point, the range is much larger than the range would be with signed fixed point with a number of integer bits. The scale is essentially built into the format, and mathematical operations need no extra passes to normalize the final result.